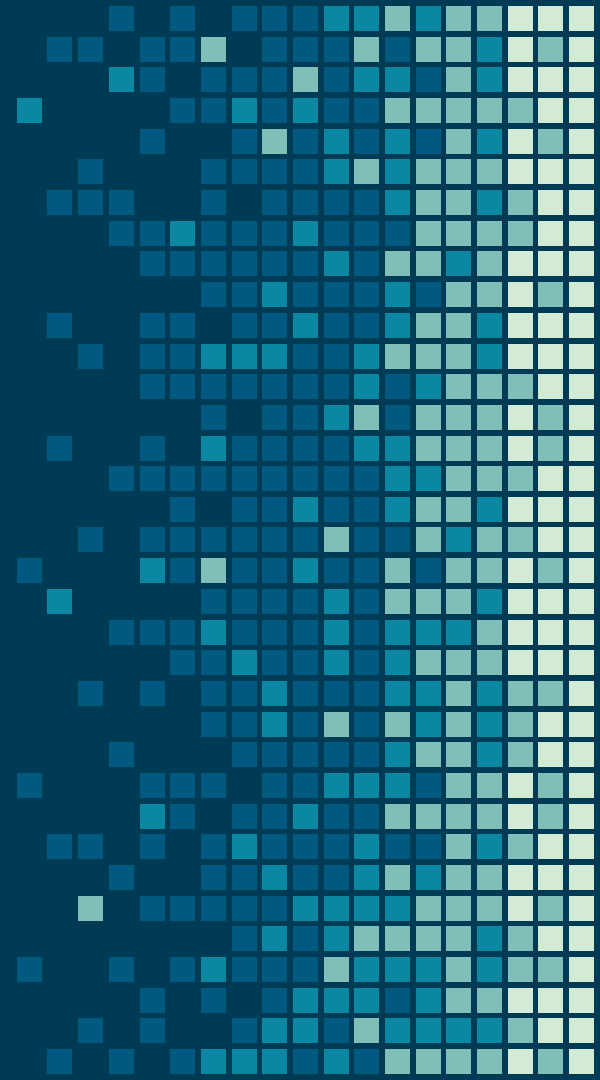
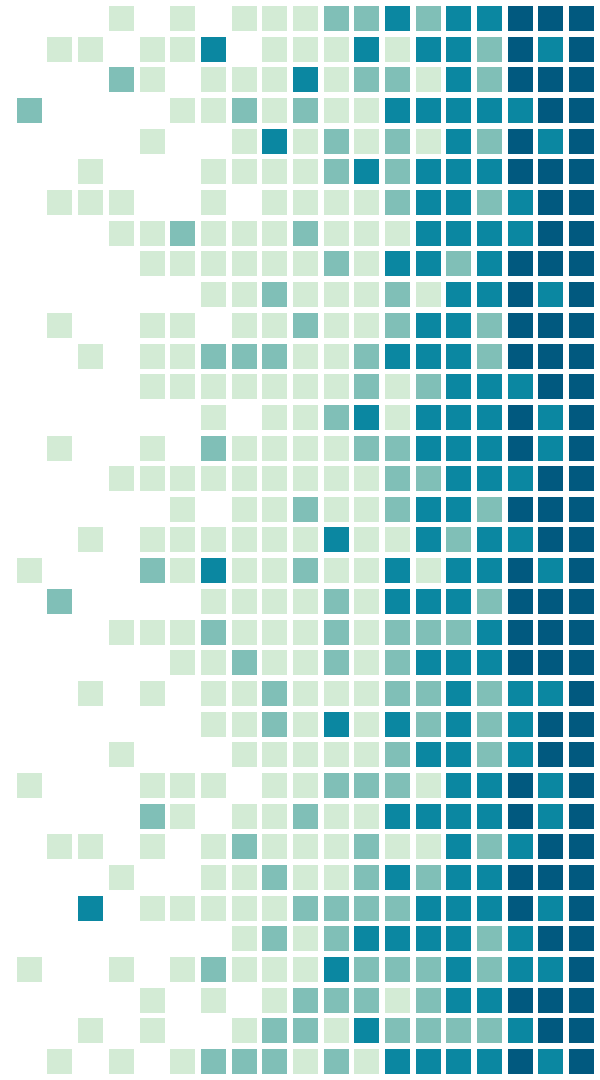


C++ from zero to hero Bonus Chapters



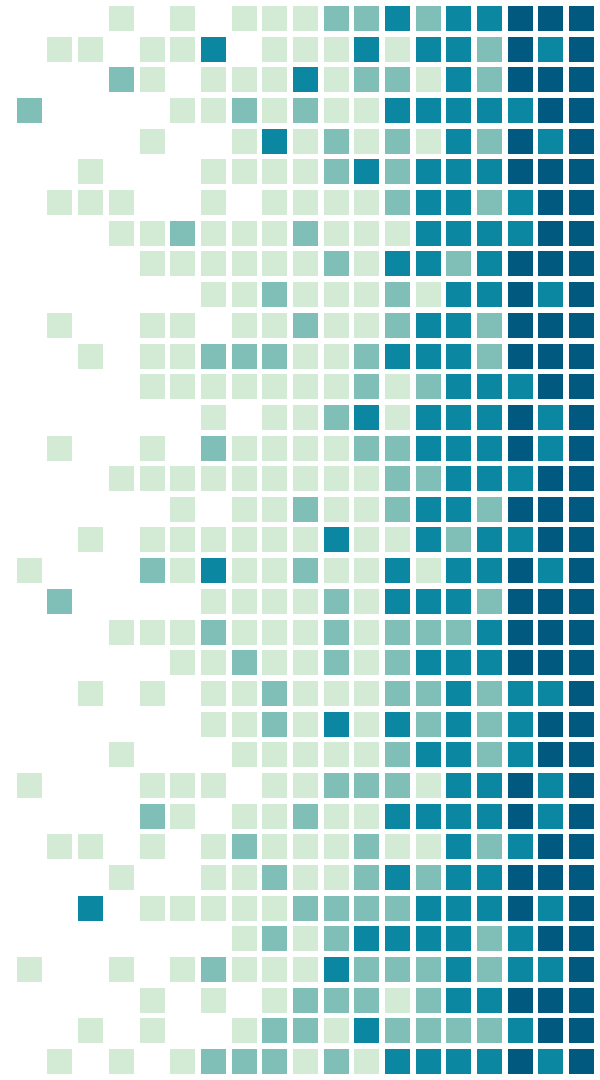


Quando inizi a studiare il C++



1.

Next-Gen Algorithms



Range-v3 / C++20 ranges

- A seguito dell'incredibile lavoro di *Eric Niebler*, il C++20 formalizza il concetto di **Range** e introduce un nuovo paradigma per lavorare sui dati
- Un range è qualsiasi tipo "iterabile" (formalmente: deve avere `begin` / `end`)

Ad esempio:

- `vector`
- `string`



Range-v3 / C++20 ranges

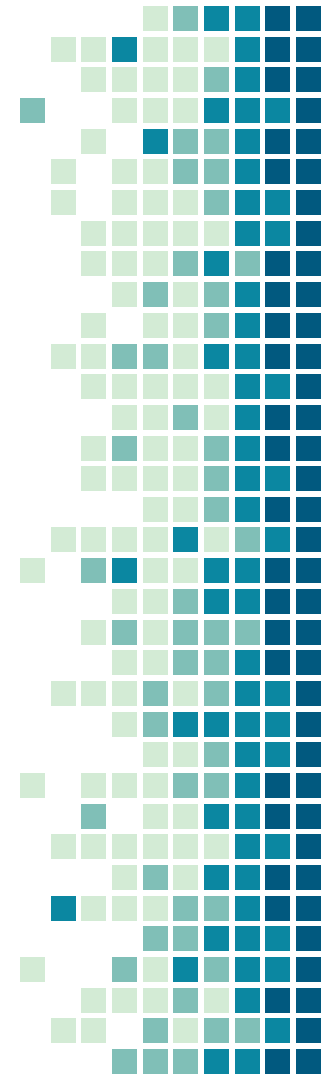
- Banalmente, tutto ciò che può essere usato in un *range-based for loop* è un range

```
for (auto i : rng)  
    // usa i
```

Range-v3 / C++20 ranges

Possibili implementazioni di range:

- Una coppia di iteratori
- Un iteratore ed un contatore
- Un iteratore ed un predicato



Range-v3 / C++20 ranges

- Gli algoritmi STL (non tutti) vengono dotati di overload che prendono un range:
- Ad esempio:

```
vector<int> v = {...};  
sort(v);  
auto where = find(v, 10);
```

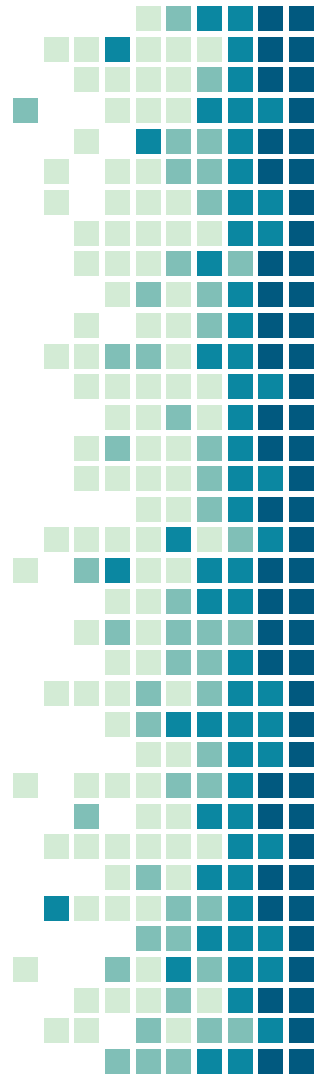
Range-v3 / C++20 ranges

La vera rivoluzione è il concetto di **view**:

- wrapper che trasforma qualcosa in un range e ne nasconde la logica iterativa
- può **riferirsi** a dei dati ma non li può modificare
- **descrivere** le operazioni da effettuare
- può essere composta in una **pipeline**
- ha una valutazione **lazy**

Range-v3 / C++20 ranges

```
vector v = {1,2,3,4};  
auto view = views::filter(v, [](auto i) {  
    return i%2==0;  
});  
for (auto i : view)  
    cout << i << "\\n";
```



Range-v3 / C++20 ranges

```
vector v = {1,2,3,4,5,6,7,8};  
auto rng = views::filter(v, [](auto i) {  
    return i%2==0;  
    })  
    | views::take(3);  
for (auto i : rng )  
    cout << i << "\\n";
```

- la view viene valutata solo quando ci si itera sopra

Range-v3 / C++20 ranges

```
auto rng =    views::iota(0) // [0,1,2,3,...]
              | views::filter(v, [](auto i) {
                  return i%2==0;
              }) | views::take(3);
for (auto i : rng )
    cout << i << "\\n";
```

- la view viene valutata solo quando ci si itera sopra

Range-v3 / C++20 ranges

- Per modificare i dati possiamo ricorrere agli algoritmi oppure ad un altro concetto chiamato **action**

```
vector v = {10, 22, 10, 5, 20, 6, 5, 1, 4, 10};  
v |= actions::sort | actions::unique;
```

- In questo caso la valutazione **non è lazy** ma ogni blocco della pipeline consuma i dati e poi li restituisce al blocco successivo

Esempi

Problemino di riscaldamento per interview tecnica:

Data una stringa di questo tipo: "aaaabbbcca"

Tornare in output la "compressione" dei caratteri uguali contigui, ad esempio:

[("a", 4), ("b", 3), ("c", 2), ("a", 1)]

```
std::string input = "aaaabbbcca";
auto output = input | views::group_by(std::equal_to<>{}) // [a,a,a,a],[b,b,b],[c,c],[a]
                    | views::transform([](auto subr) {
                        return std::make_pair(front(subr), size(subr));
                    });
for (auto [letter, size] : output)
    std::cout << letter << "," << size << "\n";
```

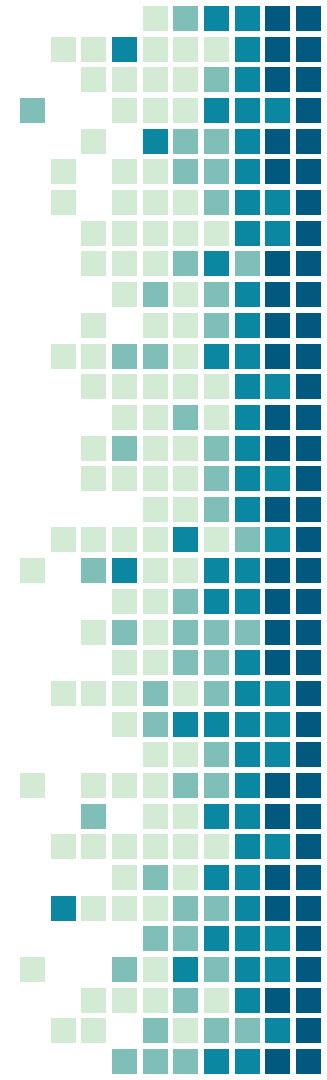
Esempi

Matrix access (courtesy of WalletFox)

```
std::vector<std::vector<int>> m = { {1,2,3}, {4,5,6}, {7,8,9} };
auto nr = distance(m); // number of rows = 3
auto nc = distance(front(m)); // number of columns = 3
std::cout << "rows= " << nr << " cols=" << nc << "\n";

auto allRows = m | views::join; // [1,2,3,4,5...]
auto c0 = allRows | views::drop(0) | views::stride(nc); // [1,4,7]
auto c2 = allRows | views::drop(2) | views::stride(nc); // [3,6,9]
auto diagonal = allRows | views::stride(nc + 1); // [1,5,9]

std::cout << "c0: " << c0 << "\n";
std::cout << "c2: " << c2 << "\n";
std::cout << "diagonal: " << diagonal << "\n";
```



Esempi

FizzBuzz rivisitato (courtesy of WalletFox)

```
std::array<std::string,3> fizz {"","","Fizz"};
std::array<std::string,5> buzz {"","","","","Buzz"};
auto r_fizzes = fizz | views::cycle; // [ , ,Fizz, , ,Fizz...]
auto r_buzzes = buzz | views::cycle; // [ , , , ,Buzz, , , , ,Buzz...]
auto r_fizzbuzz = views::zip_with(std::plus{}, r_fizzes, r_buzzes); // [ , ,Fizz, ,Buzz,Fizz, , ,Fizz,...]
auto r_int_str = views::iota(1) | views::transform([](int x){ return std::to_string(x);}); // [1,2,3...]
auto rng = views::zip_with([](auto a, auto b){return std::max(a,b);}, r_fizzbuzz, r_int_str);
std::cout << (rng | views::take(20)) << "\n";
```

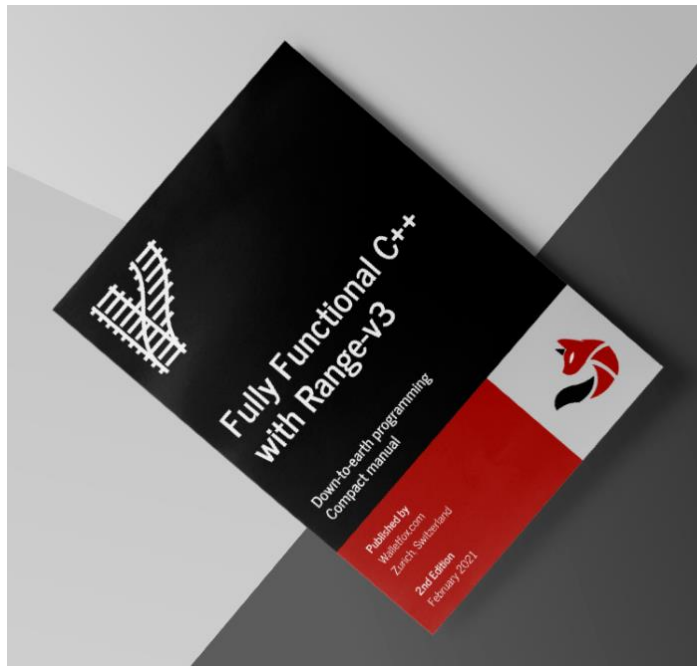


La reazione media al codice di prima

Per allenarsi...

25% sconto:

ITALIANCPP25



https://www.walletfox.com/publications_ranges.php

2.

Concorrenza & Parallelismo



C++ Concurrency Support History

- C++11 aggiunge il supporto alla **concorrenza**
(thread, future, mutex, atomic, memory model, ecc)
- C++17 aggiunge il supporto agli **algoritmi paralleli**
- C++20 aggiunge il supporto alle **coroutine**
- C++23 potrebbe aggiungere supporto agli **executor**

<thread>

```
cout << "Main thread: " << this_thread::get_id() << "\n";  
thread t1{[] {  
    cout << "hello from another thread: " <<  
        this_thread::get_id() << "\n";  
}};  
t1.join();
```

<thread>

```
vector<jthread> pool;  
for (auto i=0; i<10; ++i)  
{  
    pool.emplace_back([]{  
        // some heavy computation...  
    });  
} // jthread fa join automatico quando viene distrutto
```

<future>

```
int ComplicataFunzioneCheFaCose()
{
    this_thread::sleep_for(1s);
    return 23;
}

//...

// questo è un future<int>
auto resultWillBeHere = async(ComplicataFunzioneCheFaCose);
cout << "nel frattempo faccio altro...\n";

cout << resultWillBeHere.get() << "\n";
```

<future>

```
void ComplicataFunzioneCheFaCoseConPromise (promise<int> p)
{
    p.set_value(ComplicataFunzioneCheFaCose());
}
```

```
//...
```

```
promise<int> p;
auto alsoWillBeHere = p.get_future();
ComplicataFunzioneCheFaCoseConPromise (move(p));

cout << alsoWillBeHere.get() << "\n";
```

Parallel STL

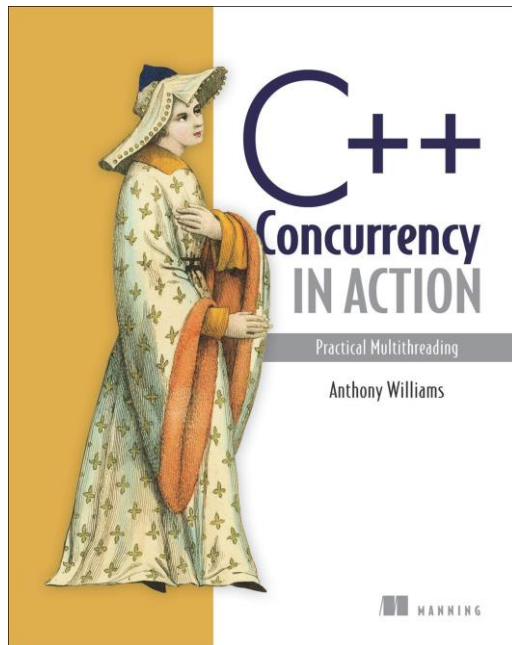
- Molti algoritmi vengono dotati di un overload che ne permette l'esecuzione attraverso **policy di parallelizzazione**:
 - `sequenced_policy`: esecuzione non parallela (à la STL "classica")
 - `parallel_policy`: esecuzione **può** avvenire su altri thread (le operazioni su tali thread non sono "interleaved")
 - `parallel_unsequenced_policy`: esecuzione **può** essere vettorizzata o distribuita su più thread (interleaved possibile)
 - `unsequenced_policy`: esecuzione **può** essere vettorizzata (e.g. SIMD)

<algorithm> + <execution>

```
// esempio di parallel STL
vector<int> v(1'000'000);
default_random_engine re;
uniform_int_distribution<int> dist(0, 1024);
generate(begin(v), end(v), [&]{
    return dist(re);
});

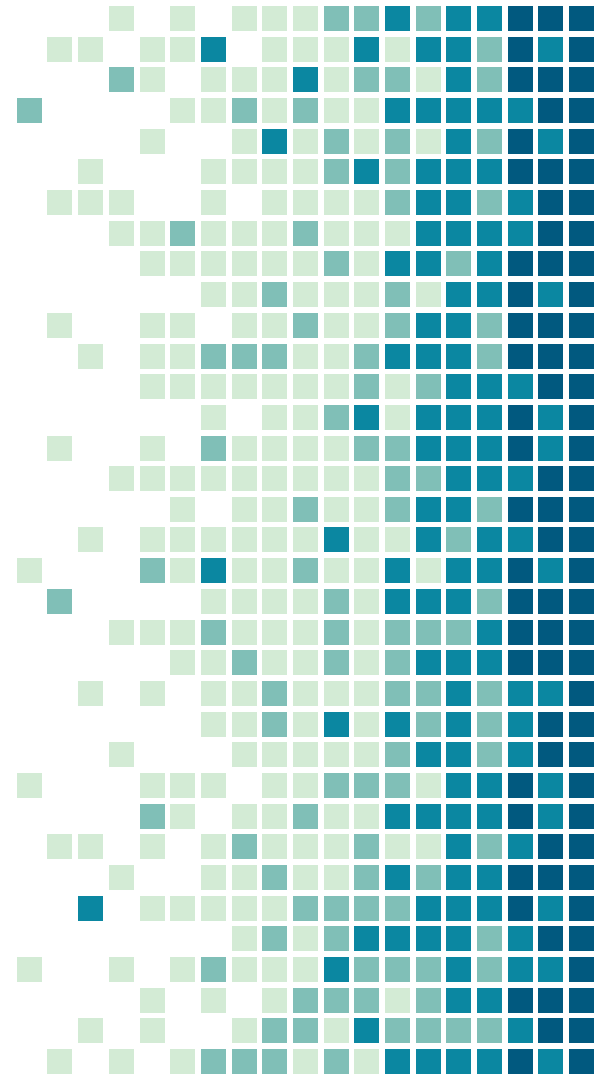
cout << *max_element(begin(v), end(v)) << "\n";
cout << *max_element(std::execution::par, begin(v), end(v)) << "\n";
```

Per approfondire



<https://www.manning.com/books/c-plus-plus-concurrency-in-action-second-edition>

3. Lifetime Patterns



Lifetime in pillole

- In C++ non c'è il garbage collector
- di default, ogni oggetto viene distrutto quando esce di scope (**lifetime automatico**)

```
void pippo() {  
    Oggetto o; // inizializzazione di o  
    o.Foo();  
    ...  
} // o viene distrutto in uscita dallo scope
```

Lifetime in pillole

- Per “estendere” il lifetime di un oggetto si può ricorrere al **lifetime dinamico**

```
Foo* pippo() {  
    Foo* o = new Foo();  
    o->Foo();  
    return foo; // compito del chiamante distruggerlo  
}
```

Lifetime in pillole

- Per “estendere” il lifetime di un oggetto si può ricorrere al **lifetime dinamico**

```
Foo* p = foo() {  
    Foo* c = new Foo();  
    c->foo();  
    return foo; // compito del chiamante distruggerlo  
}
```

VERY BAD!

Lifetime in pillole

- Un altro problema del codice di prima sorge quando un puntatore viene **copiato**:

```
Foo* f = new Foo();  
Foo* f2 = f;  
...  
delete f;  
// e se faccio delete f2 ???
```

Lifetime in pillole

- Si prova ad evitare il più possibile di gestire a mano il lifetime degli oggetti e delle risorse
- Si usano, invece, dei **wrapper**
- I wrapper usano il principio di gestione automatica della memoria, usando questa idea:
 - **Inizializzo** la risorsa “dinamica” nel costruttore
 - **La rilascio** nel distruttore

Resource Acquisition Is Initialization (RAII)

Lifetime in pillole

- Il C++ **garantisce** che il distruttore venga chiamato anche in caso di eccezione

```
{  
    ifstream file("pippo.txt"); // file viene aperto  
    string s;  
    file >> s;  
} // file viene chiuso
```

Lifetime in pillole

- Gestire il lifetime comporta anche decidere la **semantica** delle operazioni di copia (e di move)
- In generale si cerca di delegare questo compito solo a degli oggetti specifici, seguendo la filosofia della **Rule of Zero** (applicazione di [SRP](#))
- In poche parole, le nostre classi di business non dovrebbero **mai** avere a che fare col lifetime degli oggetti, ma **queste responsabilità vanno incapsulate in oggetti specifici che fanno solo quello**

“Wrapper RAI portatili”

- Ma dobbiamo riscrivere un wrapper RAI ad-hoc ogni volta che ci serve gestire lifetime dinamico?
- No!
- Oltre a tipi che già conosciamo (come i **containers**), esistono dei “wrapper RAI portatili” che fanno del lavoro per noi con una semantica ben precisa:
gli **smart pointers**

“Wrapper RAI portatili”

- Uno **smart pointer** wrappa un puntatore, ne fornisce accesso e ne gestisce automaticamente il lifetime attraverso una ben precisa semantica

```
Pippo* pippo = new Pippo();  
pippo->Foo();  
delete pippo;
```

```
unique_ptr<Pippo> pippo{new Pippo()};  
pippo->Foo();  
// distrutto in automatico
```

“Wrapper RAI portatili”

- Che succede quando uno smart pointer viene copiato?

```
Foo* f = new Foo();  
auto f2 = f;  
...  
delete f;  
// se per sbaglio faccio delete f2 ???
```

```
unique_ptr<Foo> f{ new Foo() };  
auto f2 = f; // ???
```

“Wrapper RAI portatili”

- Che succede quando uno smart pointer viene copiato?
- Dipende dalla semantica...
- Il codice di prima non compila!
- `unique_ptr` è come il proprietario di un'auto:
 - è l'**unico** proprietario
 - può trasferire la proprietà a qualcun altro
(e non costa 400€)



```
unique_ptr<Foo> f{ new Foo() };  
auto f2 = move(f);  
// qui f non ha più niente dentro
```

“Wrapper RAI portatili”

- Esistono altri smart pointer?
- Sì, nella libreria standard ce ne sono altri due ma li teniamo fuori dai contenuti per non aggiungere troppa entropia...
- Vale la pena solo menzionare `shared_ptr` che modella una semantica di **reference counting** sul puntatore che wrappa

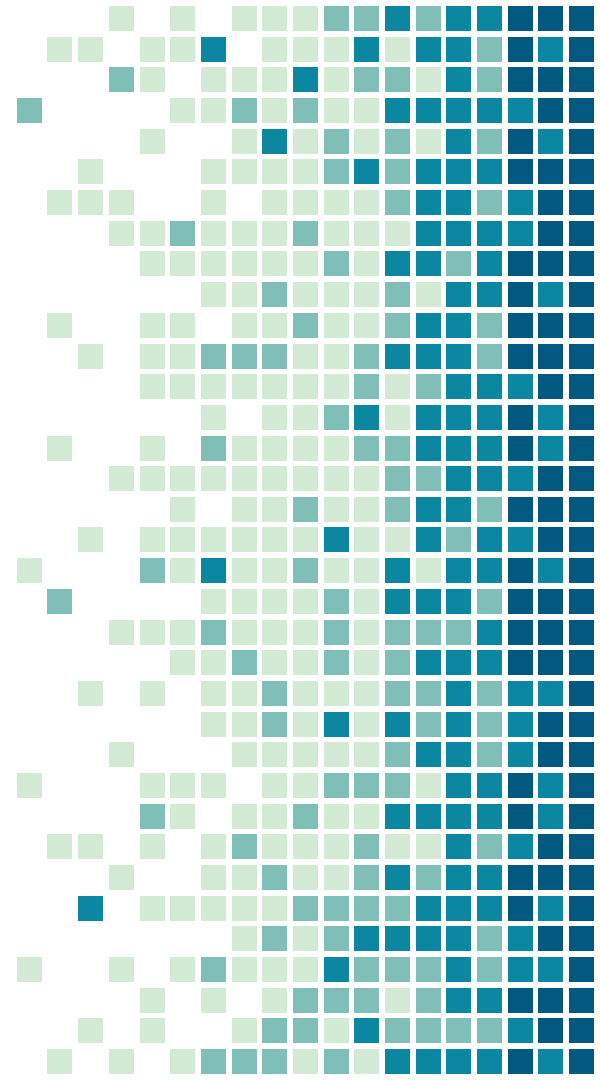
“Come creo un vector di oggetti astratti?”

```
vector<unique_ptr<ILogger>> loggers;  
loggers.push_back(make_unique<ConsoleLogger>());  
loggers.push_back(make_unique<FileLogger>());  
loggers.push_back(make_unique<DebugViewLogger>());  
//...  
loggers[0]->Log("ciao");
```

Tuttavia esistono diverse alternative a questo tipo di design, sia come stile di programmazione che come paradigma...

Ad esempio: <https://www.youtube.com/watch?v=bIhUE5uUFOA>

4. Programmazione a Compile-Time



Programmazione generica

- Funzioni e classi sono scritte usando dei tipi che vengono specificati in seguito
- Un modello primitivo di programmazione generica è costituito dall'overloading

```
int add(int a, int b) { return a + b; }
```

```
float add(float a, float b) { return a + b; }
```

Programmazione generica

- Tuttavia, nell'esempio precedente non potrei aggiungere "automaticamente" l'overload per `std::string`, nonostante questo tipo supporti l'operatore **plus (+)**
- Servirebbe un modo per farlo **generare** in automatico dal compilatore usando una sorta di **template**

```
Type add(Type a, Type b) { return a + b; }
```

Template in C++

- Possiamo vedere un template come un **generatore** di codice che il compilatore può usare per **produrre** famiglie di classi e funzioni dove certi tipi vengono **sostituiti** in base all'utilizzo che se ne fa (tale sostituzione è fatta a compile-time)

```
template<typename T>
T add(T a, T b) { return a + b; }

add(1, 1); // compilatore produce add<int>
add(string{"ciao"}, string{"mondo"}); // produce add<string>
```

Un altro paradigma di programmazione

- Mentre nel paradigma Object-Oriented i tipi vengono discriminati in base alla loro **interfaccia pubblica** (e.g. ILogger), con i template **contano solamente le operazioni** che vengono fatte su di essi

```
template<typename T>
void DuckThis(T& duck) {
    duck.Duck();
}
```

```
void DuckThis(IDuck& duck) {
    duck.Duck();
}
```

Generic Programming in C++

- Con i template si possono fare delle cose incredibili e di una complessità disarmante...
- Tuttavia, il linguaggio e la libreria *permettono* di nascondere questa complessità...

```
cout << "ciao";  
cout << 10;  
cout << mioTipoCheSupportaOutOperator;
```

Generic Programming in C++

- Il mindset della generic programming è che **molte decisioni possono essere prese a tempo di compilazione**, influenzando sul codice che viene generato
- Ad esempio, l'algoritmo `std::copy` riesce a copiare due range di qualsiasi tipo
- In alcuni casi la copia è fatta in blocco (e.g. `memcpy`)
- Questa possibilità viene "decisa" a compile-time

Generic Programming in C++

- Nel caso di `std::copy`, gli implementatori della libreria possono **ispezionare** il tipo del range passato e, nel caso di un “oggetto semplice” (e.g. un POD) optare per un `memcpy` anziché una copia elemento per elemento

```
vector<int> v1 = {1,2,3,4};  
vector<int> v2(4);  
copy(begin(v1), end(v1), begin(v2));
```


Generic Programming in C++

```
template<typename T>
void copy_naive_impl(vector<T>& src, vector<T>& dst, true_type) {
    memcpy(dst.data(), src.data(), sizeof(T) * src.size());
}

template<typename T>
void copy_naive_impl(vector<T>& src, vector<T>& dst, false_type) {
    std::copy(begin(src), end(src), begin(dst));
}

template<typename T>
void copy_naive(vector<T>& src, vector<T>& dst) {
    copy_naive_impl(src, dst, bool_constant<is_trivially_copyable_v<T>>{});
}
```

Esempio mooolto semplice di **tag dispatching**



La reazione media al codice di prima



Quando ci prendi gusto coi template

Generic Programming in C++

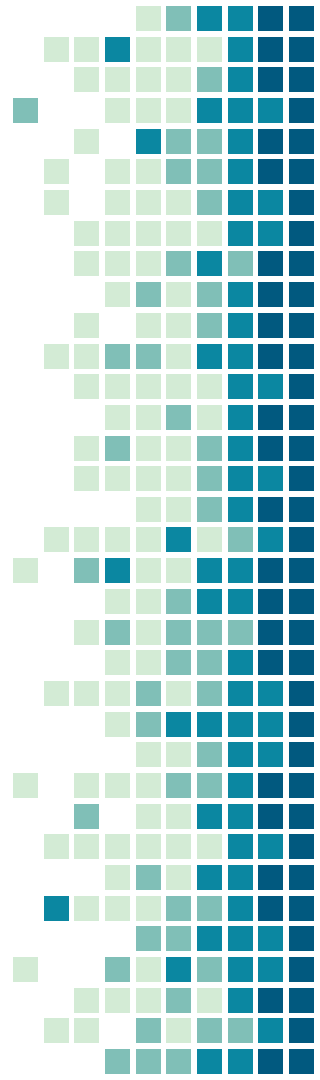
- Decenni di evoluzione hanno reso il **C++ uno dei linguaggi più potenti** per scrivere **codice generico**
- Contestualmente, diversi idiomi e "trucchi" possono ritenersi superati dalle recenti evoluzioni del linguaggio e della libreria standard

Qualche altro esempio

```
template<typename T, typename Action>
void Iterate(const vector<T>& data, Action action) {
    if constexpr (is_invocable_v<Action, T>) {
        for_each(begin(data), end(data), action);
    }
    else {
        auto idx = 0;
        for (const auto& c : data) {
            action(c, idx++);
        }
    }
}

vector<string> v1 = {"ciao", "come", "stai"};
Iterate(v1, [](auto i, auto index) {
    cout << i << " " << index << "\n";
});

Iterate(v1, [](auto i) {
    cout << "v1[" << index << "]=" << i << "\n";
});
```





Quando metti in produzione una
nuova feature oscura del linguaggio
per essere cool